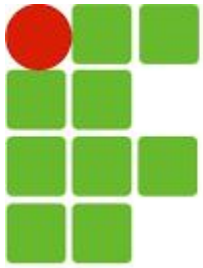


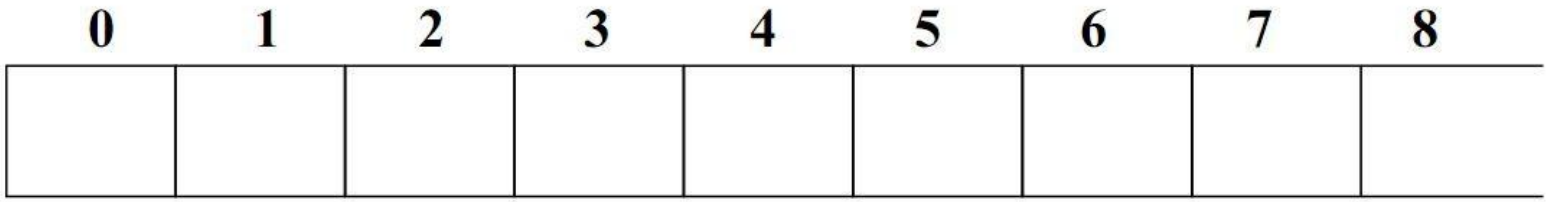
INSTITUTO FEDERAL
GOIÁS
Câmpus Anápolis

Ponteiros & Alocação dinâmica

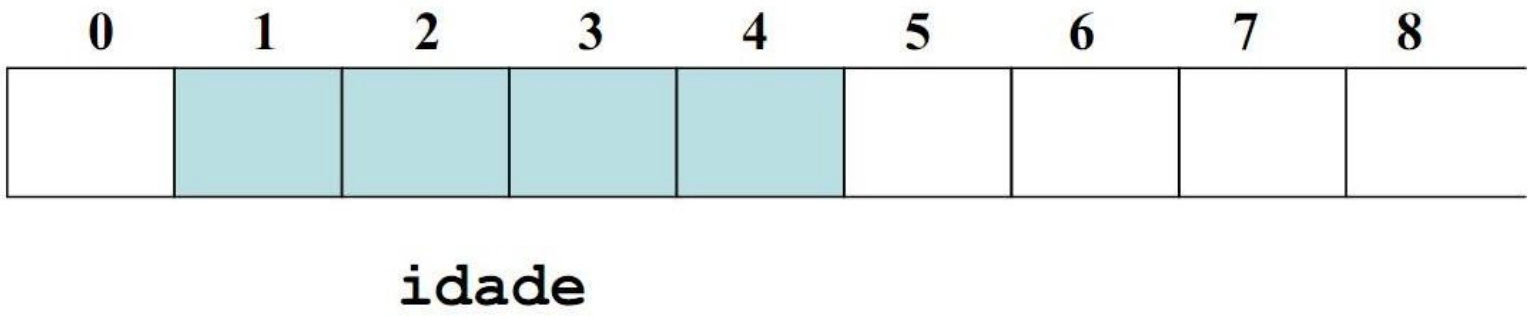
Sergio Canuto
Sergio.canuto@ifg.edu.br

“Memória?”

Uma memória é uma seqüência de células de armazenamento (ou posições)



“ Uma variável é um aglomerado de uma ou mais células de memória ” .



Variável

Atributos de uma variável

–nome: seqüência de caracteres utilizada para identificar a variável;

– tipo: é o tipo dos dados que serão armazenados na variável;

– conteúdo: é o valor armazenado na variável;

– **endereço**: é a localização (posição) da variável na memória;

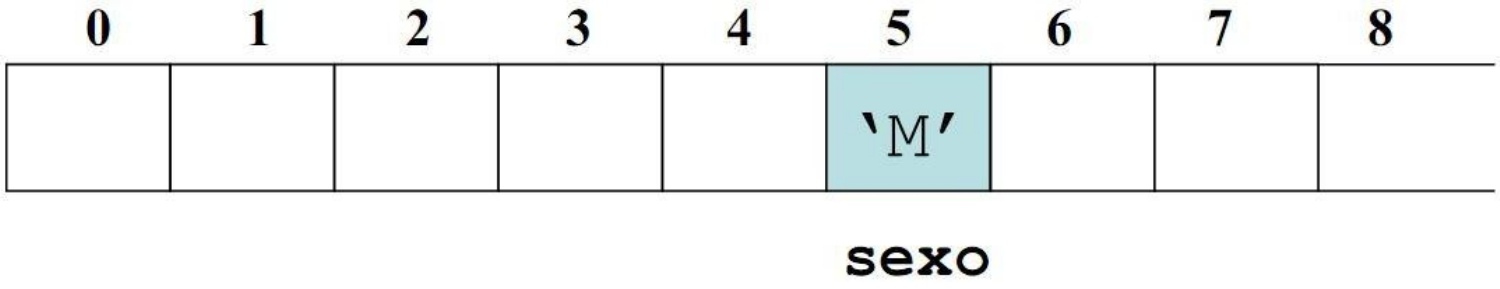

```
var  nome: caractere
     idade: inteiro
     ativo: logico

                    //atribuindo o valor "Ana Maria" a um endereço de memória
                    @#$1208DFE <- "Ana Maria";
                    //atribuindo o valor 27 a um endereço de memória
                    @#$1118DCC = 27;
```

Variável

– char sexo = ‘M’;

- Nome da variável: sexo
- Tipo da variável: char
- Conteúdo da variável: ‘M’
- Endereço da variável: 5



Variável

– int idade = 31;

- Nome da variável: idade
- Tipo da variável: int
- Conteúdo da variável: 31
- Endereço da variável: 1

Variável

– int idade = 31;

- Nome da variável: idade
- Tipo da variável: int
- Conteúdo da variável: 31
- Endereço da variável: 1

Variáveis do tipo ponteiro

- Ponteiros, como o próprio nome diz, é um tipo de variável que aponta para uma posição de memória (de um tipo qualquer).
  - ex. de tipo: Ponteiro para inteiros, ponteiro para char, etc.

• Um ponteiro guarda o endereço de memória de uma variável.

•Sintaxe: Variáveis que sãoponteiros sãorepresentadasda seguinte forma:

tipo_dado_apontado *nome_variavel;

- **int *ptr1;**
- **char *ptr2;**

Variáveis do tipo ponteiro

Utilizado para:

- Manipulação de vetores/matrizes com eficiência;
- Passar valores e mudar valores dentro de funções;
- Manipular arquivos;
- Aumento de eficiência para algumas rotinas;
- Possibilitar a alocação dinâmica de memória.

Variáveis do tipo ponteiro

Operadores de Endereço

- O **&** ("endereço de") antes de um nome de variável qualquer retorna o endereço desta variável.

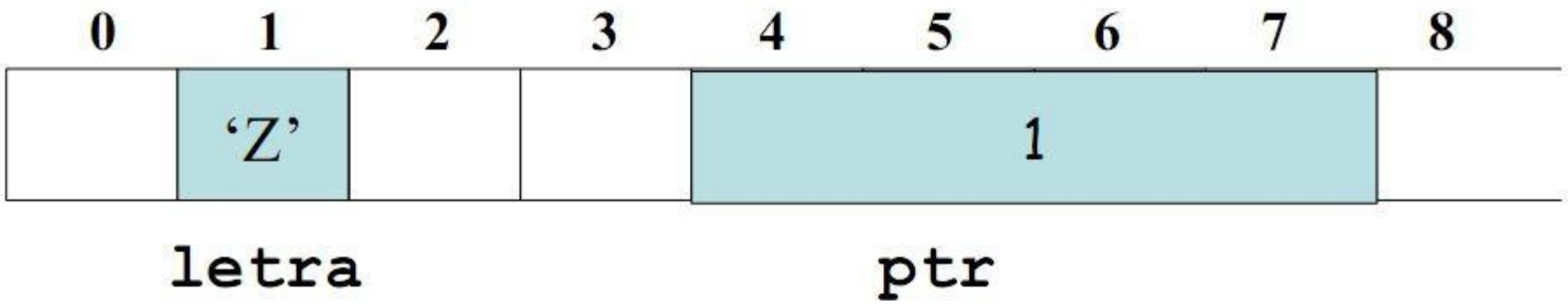
- O ***** ("conteúdo de") antes de um nome de variável.

- O ponteiro retorna conteúdo armazenado na posição de memória.
- O endereço retorna o lugar da memória onde se encontra o conteúdo.

Variáveis do tipo ponteiro

char letra = ‘Z’;
char *ptr = &letra;

- Nome da variável: ptr
- Tipo da variável: char *
- Conteúdo da variável: 1
- Endereço da variável: 4



Variáveis do tipo ponteiro

```
#include<stdio.h>
#include<stdlib.h>

void main(){
int x = 50;
int * ptr;
printf ("%i\n",x);      /*exibe o conteúdo da variável x
(50)*/
printf ("%p\n",&x); /*exibe o endereço da variável x*/
ptr = &x; /*armazena em ptr o endereço de x*/
printf ("%p\n",ptr); /*exibe o conteúdo da variável
ptr*/
printf ("%p\n",&ptr); /*exibe o endereço da variável
ptr*/
printf ("%i\n",*ptr); /*exibe o conteúdo da variável x*/
}
```

Variáveis do tipo ponteiro

```
void main () {
int x;
int *ptr;
printf("Digite um valor para x:");
scanf("%i", &x);
ptr = &x;
*ptr = *ptr + 1;
printf("O valor de x é %i", *ptr);
printf("O endereço de x é %i", ptr);
  }
```

Variáveis do tipo ponteiro

***p** ⇒ representa o conteúdo da variável apontada.

p ⇒ representa o endereço de memória da variável apontada.

Variáveis do tipo ponteiro

Observação
– Quando queremos indicar que um ponteiro está "vazio", ou seja, **não contém um endereço de uma variável**, atribuímos a ele o valor **NULL**.

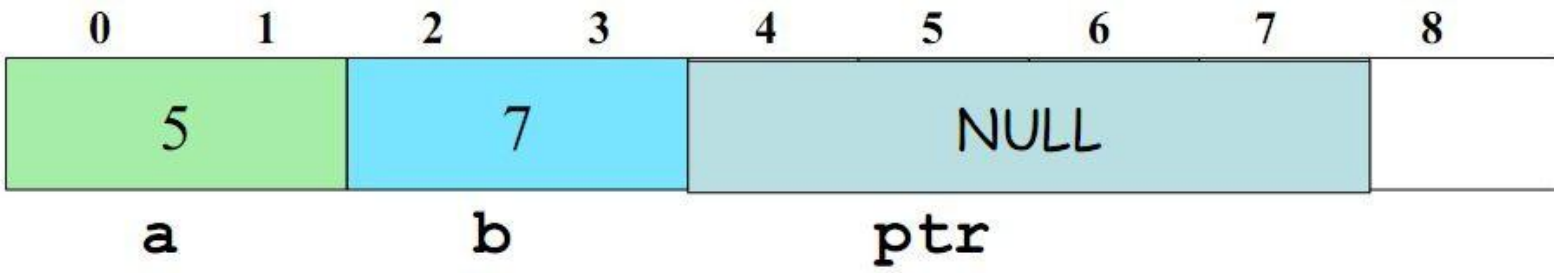
- **NULL**: Endereço de memória 0 (zero). Esta posição de memória não é utilizada para armazenar dados.

– Exemplo:
ptr = NULL; /* inicializa o ponteiro ptr*/

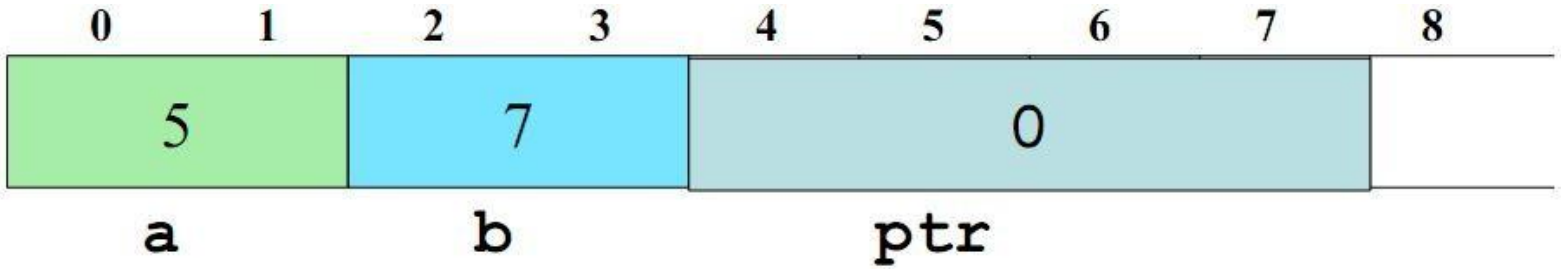
Variáveis do tipo ponteiro

int a = 5, b = 7; (neste exemplo, um int ocupa 2bytes na memoria, e não 4, como em outros exemplos).

int *ptr = NULL;



ptr = &a;



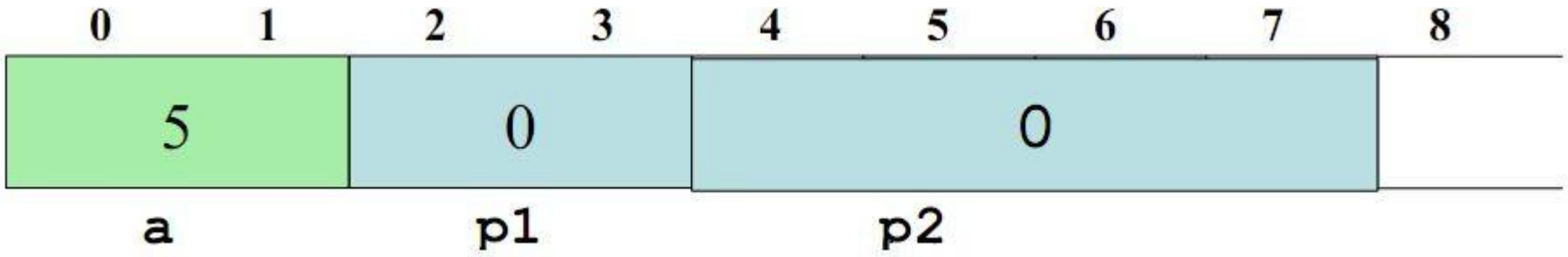
Variáveis do tipo ponteiro

Atribuição

– Podemos atribuir o conteúdo de um ponteiro a outro. Dessa forma, teremos dois ponteiros referenciando o mesmo endereço de memória.

• Exemplo:

```
int a = 5, *p1, *p2;
p1 = &a;
p2 = p1;
```



Aritmética de ponteiros

Incremento e Decremento

- Um ponteiro pode ser incrementado como qualquer outra variável

- Se ptr é um ponteiro para um determinado tipo, quando ptr é incrementado, por exemplo, de uma unidade, o endereço que passa a conter é igual ao endereço anterior de ptr + sizeof(tipo) para que o ponteiro aponta

- A mesma situação ocorre para a operação de decremento, onde o endereço que passa a conter é igual ao endereço anterior de ptr - sizeof(tipo) para que o ponteiro aponta

Aritmética de ponteiros

```
int main(int argc, char **argv) {
int x = 5, *px = &x;
double y = 5.0, *py = &y;

 printf("%i %i\n", x, px);
 printf("%i %i\n", x+1, px+1);
 printf("%f %i\n", y, py);
 printf("%f %i\n", y+1, py+1);
 printf("%i %i\n", x, px);
 printf("%i %i\n", x-1, px-1);
 printf("%f %i\n", y, py);
 printf("%f %i\n", y-1, py-1);
 getchar();
return 0;
}
```

```
5 2293524
6 2293528
5.000000 2293512
6.000000 2293520
5 2293524
4 2293520
5.000000 2293512
4.000000 2293504
```

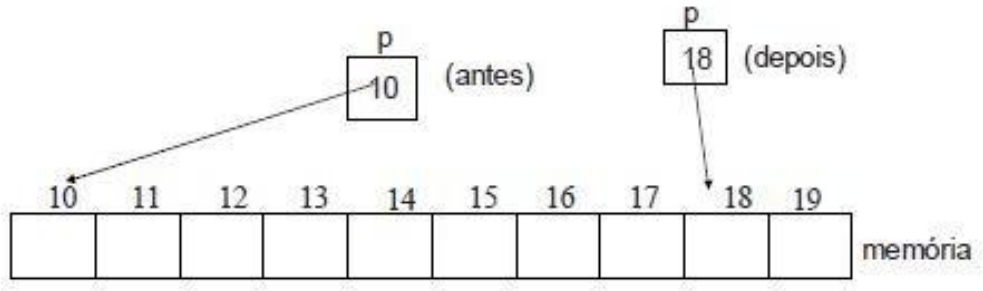
Aritmética de ponteiros

- **Adição: somar um inteiro a um ponteiro**

•Exemplo:

float *p;

p = p + 2;



Fator de escala: é o tamanho (número de bytes) do objeto apontado.

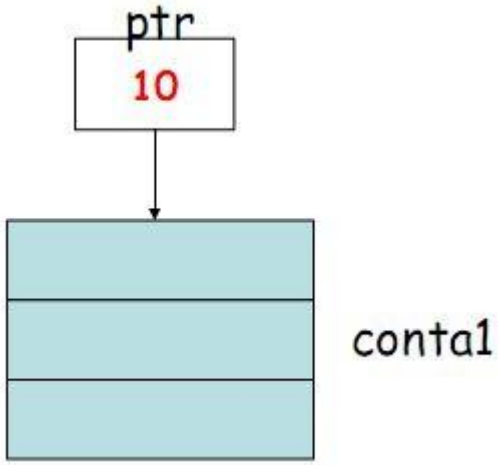
- Exemplo: fator de escala de uma variável do tipo float: 4

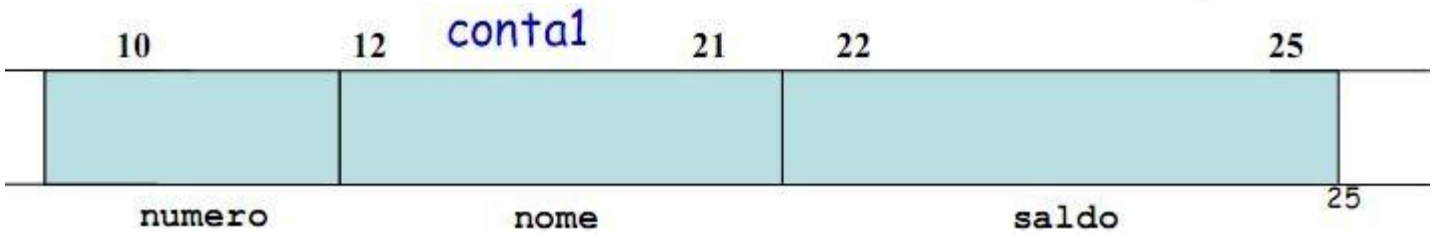
p = p + 2 significa p = p + 2 * fator de escala

Ponteiro para estrutura

Como os outros tipos do C, as estruturas podem ser referenciadas usando ponteiros.

```
typedef struct {
int numero;
char nome[10];
float saldo;
} conta;
conta conta1;
conta *ptr;
ptr = &conta1;
```





Ponteiro para estrutura

Há duas formas para recuperar os valores de uma estrutura usando o ponteiro:

– Se st é uma estrutura e ptr é um ponteiro para st, para referenciar a estrutura usamos:

(*ptr).elemento ou ptr->elemento

Ponteiro para estrutura

```
conta conta1, *ptr;
conta1.num_conta = 2;
strcpy(conta1.nome, “Maria”);
conta1.saldo = 482.25;
ptr = &conta1;
printf(“Numero = %i\n”, (*ptr).num_conta);
printf(“Nome = %s\n”, (*ptr).nome);
pritnf(“Saldo = %f\n”, (*ptr).saldo);
/* ou */
printf(“Numero = %i\n”, ptr->num_conta);
printf(“Nome = %s\n”, ptr->nome);
pritnf(“Saldo = %f\n”, ptr->saldo);
```

```
typedef struct {
int num_conta;
char nome[10];
float saldo;
} conta;
```

Ponteiro para estrutura

```
typdef struct{
    char titulo [40];
    float preco;
} livro ;

void main()
{
    livro liv, *ptr;
    gets (liv.titulo);
    scanf("%f", &liv.preco);  ptr = &liv;
    ptr->preco = ptr->preco * 0.1;
    printf("O preço de %s eh %f.", pt->titulo,
    pt->preco);
}
```

Ponteiro e Vetor

O nome de um vetor corresponde ao endereço do seu primeiro elemento, isto é, se v for um vetor então v é igual ao &v[0].

Embora o nome de um vetor seja um ponteiro para o primeiro elemento do vetor, esse ponteiro não pode ser alterado durante a execução do programa a que pertence.

Ponteiro e Vetor

Existem duas formas de colocar um ponteiro apontando para o primeiro elemento de um vetor:

```
int v[3] = {10, 20, 30};
int * ptr;

ptr = &v[0]; // ou  ptr = v;
```

Ponteiro e Vetor

Ao contrário de v, que é um vetor (ponteiro constante associado à sua própria memória), ptr é um ponteiro puro, e portanto pode receber endereços de diferentes elementos de um vetor.

```
int v[3] = {10, 20, 30};
int *ptr;
ptr = v;
printf("%i %i", v[0], *ptr); /* 10    10 */
ptr = &v[2]
printf("%i %i", v[2], *ptr); /* 30    30 */
```

Ponteiro e Vetor

Outra sintaxe

–Colocamos em c o endereço do terceiro elemento de vetor:

char vetor[5] = { ‘a’, ‘e’, ‘i’, ‘o’, ‘u’ };

char *c;

c = &vetor[2];

– Portanto:

c[0] = ‘i’; c[1] = ‘o’ e c[2] = ‘u’.

–Se tivéssemos feito c = &vetor[3], então: c[0] = ‘o’ e c[1] = ‘u’.

Ponteiro e Vetor

Usando aritmética de ponteiros, como acessar os valores do vetor usando a variável ponteiro c ?

```
#include <stdio.h>
void main(){
 int i;
 char vetor[5] = { 'a', 'e', 'i', 'o', 'u' };
 char *c;
 c = vetor;
 for (i = 0; i < 5; i++) {
 printf("\n%c ", c[i]);     /* ou */
 printf("%c ", *(c + i));
}
getchar();
}
```

Ponteiro e Vetor

Resumo

– Quando estamos manipulando ponteiros para vetores:

*c ⇒ representa o conteúdo da variável.

*(c + i) ⇒ representa o i-ésimo elemento do vetor referenciado pelo ponteiro.

c[i]⇒ também representa o i-ésimo elemento do vetor referenciado pelo ponteiro.

c ⇒ representa o endereço de memória da variável.

Ponteiro e Vetor

Exercício:

– Dadas as variáveis float vet_notas[5] e float *pont_notas, imprimir a primeira e a quinta nota usando as duas variáveis.

Ponteiro e Vetor

Exercício:

– Dadas as variáveis float vet_notas[5] e float *pont_notas, imprimir a primeira e a quinta nota usando as duas variáveis.

Passagem de Vetores para funções

Sempre que invocamos uma função e lhe passamos um vetor como parâmetro, essa na realidade não recebe o vetor na sua totalidade, mas apenas o endereço inicial do vetor, pois estamos passando v que é igual a &v[0].

Se passarmos um endereço, então a variável que recebe terá que ser um ponteiro para o tipo dos elementos do vetor.

Por essa razão é que no cabeçalho de uma função que recebe um vetor como argumento aparece um ponteiro recebendo o respectivo parâmetro.

Passagem de Vetores para funções

```
#include <stdio.h>
void exibirVetor( int * ptr, int tam ){
 int i;
 for (i = 0; i < tam; i++) {
 printf("%i ", *(ptr + i));
}
}
void main(){
int vetor[] = {1, 2, 3, 4, 5};

exibirVetor(vetor, 5);
getchar();
}
```

Ponteiro e Vetor

Conclusão

- Variáveis para vetores podem ser declaradas como sendo apontadores, pois os elementos do vetor, individualmente, têm o mesmo tratamento independente se a variável que os armazena é um vetor ou um apontador.

char vetor[5] equivale a char *vetor;

- Versões com ponteiros são mais rápidas!

Ponteiro para Ponteiro

Uma vez que os ponteiros ocupam espaço em memória, é possível obter a sua posição através do operador de endereço **&**

Pergunta: Se quisermos armazenar o endereço de um ponteiro, qual o tipo da variável que irá recebê-lo?

Ponteiro para Ponteiro

Resposta:

- Suponha uma variável do tipo int chamada x

–Se quisermos armazenar seu endereço, declaramos um ponteiro para o tipo da variável (int), isto é, colocamos um asterisco entre o tipo da variável para que queremos apontar e o nome do ponteiro

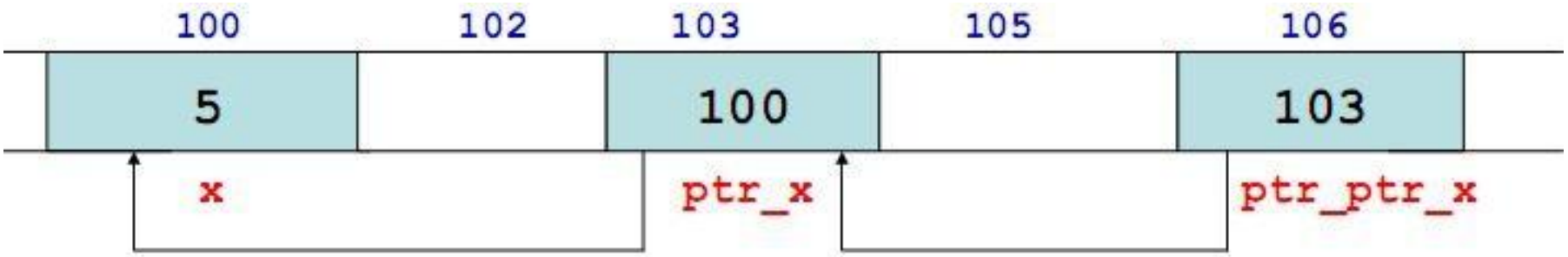
```
int *ptr_x;
```

–Se quisermos armazenar o endereço desse ponteiro, seguimos exatamente os mesmos passos

```
int **ptr_ptr_x;
```

Ponteiro para Ponteiro

```
int x = 5;
int * ptr_x;
int ** ptr_ptr_x;
/* Carga inicial dos ponteiros */
ptr_x = &x;
ptr_ptr_x = &ptr_x;
```



Função: Passagem por valor/parâmetro vs Passagem por referência/ponteiro

```c
#include <stdio.h>
//função que soma 10 ao valor recebido
void soma10(int x){
 x = x + 10;
 printf("Valor de x apos a soma = %d \n",x);
 return;
}
void soma10p(int *x){
 *x = *x + 10;
 printf("Valor de x apos a soma = %d \n",*x);
 return;
}

int main(void){
 int numero;
 printf("Digite um numero: ");                    //(ex.: 11)
 scanf("%d", &numero);
 printf("O numero digitado foi: %d \n",numero);
 soma10(numero); //chamada da função
 printf("Agora o numero vale: %d \n",numero);

 soma10p(&numero); //chamada da função com ponteiro como parâmetro
 printf("Agora o numero vale: %d \n",numero);
 return 0;
}
```

Função: Passagem por valor/parâmetro vs Passagem por referência/ponteiro

```c
#include <stdio.h>
//função que soma 10 ao valor recebido
void soma10(int x){
 x = x + 10;
 printf("Valor de x apos a soma = %d \n",x);
 return;
}
void soma10p(int *x){
 *x = *x + 10;
 printf("Valor de x apos a soma = %d \n",*x);
 return;
}

int main(void){
 int numero;
 printf("Digite um numero: ");                    //(ex.: 11)
 scanf("%d", &numero);
 printf("O numero digitado foi: %d \n",numero);    //11
 soma10(numero); //chamada da função
 printf("Agora o numero vale: %d \n",numero);     //11

 soma10p(&numero); //chamada da função com ponteiro como parâmetro
 printf("Agora o numero vale: %d \n",numero);     //21
 return 0;
}
```

ALOCAÇÃO DINÂMICA

Memória

Alocação

- Processo de vinculação de uma variável de programa a uma célula de memória de um pool de memória disponível

Desalocação

- Processo de devolução de uma célula de memória desvinculada de uma variável ao pool de memória disponível

Memória

- Alocação dinâmica:

- Espaço de memória é requisitada em tempo de execução e permanece até que seja explicitamente liberado.
  - Espaço alocado e não liberado explicitamente, será automaticamente liberado quando ao final da execução

Variáveis de Alocação Dinâmicas

A alocação de uma variável é feita por uma função chamada alocadora que retorna o endereço da variável heap alocada.

Sintaxe da função malloc do C:

void * malloc (size_t n_bytes)

Variáveis de Alocação Dinâmica

Malloc

– recebe como parâmetro o número de bytes que se deseja alocar

– retorna um ponteiro genérico para o endereço inicial da área de memória alocada, se houver espaço livre:

- ponteiro genérico é representado por void*
- ponteiro é convertido automaticamente para o tipo apropriado
- ponteiro pode ser convertido explicitamente
- retorna um endereço nulo, se não houver espaço livre:
- representado pelo símbolo NULL

Variáveis de Alocação Dinâmica

Função malloc do C

– Exemplo:

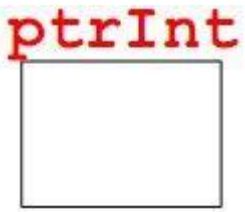
```
void main () {
int *ptrInt;
...
ptrInt = malloc(sizeof(int));
...
}
```

Cria uma variável do tipo int e coloca seu endereço no ponteiro ptrInt.

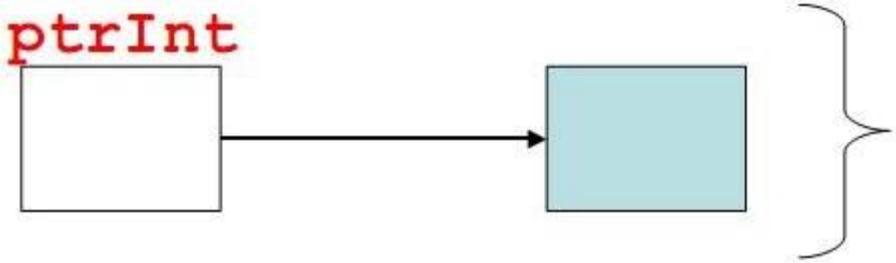
Variáveis de Alocação Dinâmica

Exemplo da função malloc do C

(i) int * ptrInt;



(ii) ptrInt = malloc(sizeof(int));



Variável do tipo int, anônima, alocada dinamicamente.

Variáveis de Alocação Dinâmica

Exemplo da função malloc do C

```
typedef struct {
  int dia, mes, ano;
} data;
data *d;
d = malloc (sizeof (data));
d->dia = 31;
d->mes = 12;
d->ano = 2008;
```

Variáveis de Alocação Dinâmica

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  // ponteiro para uma variável do tipo inteiro
  int *ponteiro;
  // aloca memória para um int
  ponteiro = malloc(sizeof(int));
  // testa se a memória foi alocada com sucesso
  if(ponteiro)
    printf("Memoria alocada com sucesso.\n");
  else
    printf("Nao foi possivel alocar a memoria.\n");
    // atribui valor à memória alocada
    *ponteiro = 45;
    // obtém o valor atribuído
    printf("Valor: %d\n\n", *ponteiro);
    // libera a memória
    free(ponteiro);
    return 0;
  }
```

Variáveis de Alocação Dinâmica

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  int i;
  // quantidade de elementos na matriz
  int quant = 10;
  // ponteiro para o bloco de memória
  int *ponteiro;
  // aloca memória para uma matriz de inteiros
  ponteiro = malloc(quant * sizeof(int));
  // testa se a memória foi alocada com sucesso
  if(ponteiro)
    printf("Memoria alocada com sucesso.\n");
    else{
      printf("Nao foi possivel alocar a
    memoria.\n");
      exit(1);
    }
// atribui valores aos elementos do array
for(i = 0; i < quant; i++){
  ponteiro[i] = i * 2;
}

 // exibe os valores
 for(i = 0; i < quant; i++){
   printf("%d     ",
   ponteiro[i]);
 }
 // libera a memória
 free(ponteiro);

 printf("\n\n");
 return 0;

}
```

Variáveis de Alocação Dinâmica

Exemplo da função malloc do C

```
int *vector = NULL; /* declaração do ponteiro */
/* alocação de memória para o vector */
vector = (int*) malloc(25 * sizeof(int));
/* altera o valor da posição dez para trinta e quatro */
vector[10] = 34;
free(vector); /* liberta a área de memória alocada */
```

Variáveis de Alocação Dinâmica

A desalocação de uma variável heap-dinâmica explícita é feita por uma função chamada desalocadora.

Sintaxe da função free do C:

void free(void *)

Variáveis de Alocação Dinâmica

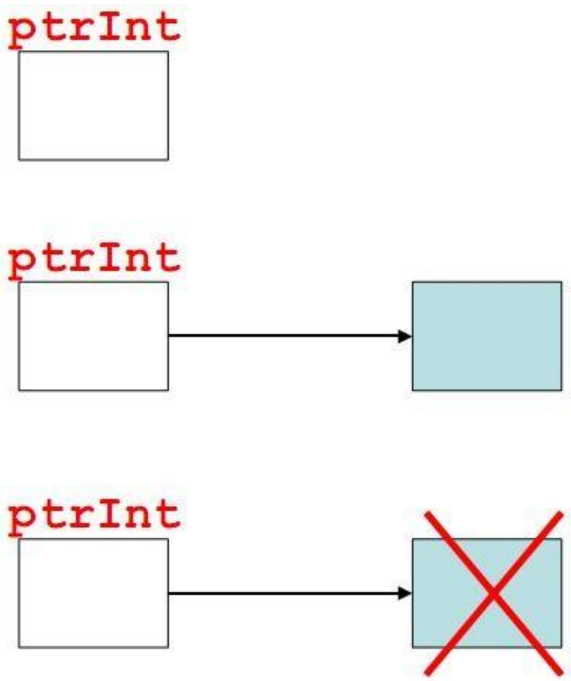
Free

- recebe como parâmetro o ponteiro da memória a ser liberada
  a função free deve receber um endereço de memória que tenha sido alocado dinamicamente

Variáveis de Alocação Dinâmica

Exemplo da função free do C:

```
void main () {
int *ptrInt;
...
ptrInt = malloc(sizeof(int));
...
free(ptrInt);
}
```



Variáveis de Alocação Dinâmica

Função free do C

– IMPORTANTE:

- A função free não desaloca o ponteiro.

Ela desaloca apenasa variável cujo endereço estava armazenado no ponteiro;

- A função free também não "limpa" o ponteiro, ele permanece com o mesmo endereço, mas este se refere a uma variável que não mais lhe pertence!

Variáveis de Alocação Dinâmica

–Uma variável heap permanece acessível enquanto houver uma variável do tipo ponteiro que armazene seu endereço.

Problemas

– Referência Perdida

–Variável Heap-Dinâmica Perdida (Lixo de Memória)

Variáveis de Alocação Dinâmica

Problemas - Referência Perdida:

– É um ponteiro que contém o endereço de um variável heap-dinâmica desalocada

– Exemplo 1:

```
void main(){
float *ptr;

…
  ptr = malloc(sizeof(float));
  …
  free(ptr);

    …
    printf("%f", *ptr); ⇐ Referência Perdida
}
```

Variáveis de Alocação Dinâmica

Problemas - Referência Perdida:

– Exemplo 2:

```
void main(){
float *ptr1, *ptr2;
…
 ptr1 = malloc(sizeof(float));
 …
 ptr2 = ptr1;
 …
 free(ptr1);
 …
 *ptr2 = 15.9; ⇐ Referência Perdida
```

Variáveis de Alocação Dinâmica

Problemas

– Heap-Dinâmica Perdida:

- É uma variável alocada porém não mais acessível.

- Também conhecida como lixo de memória

Variáveis de Alocação Dinâmica

Problemas - Heap-Dinâmica Perdida:

– Exemplo:

```
void main(){
int *ptr1, i;
...
  for(i = 0; i < 10; i++){
    ptr1 = malloc(sizeof(int)); ⇐ cria lixo
     *ptr1 = i;
   }
...
}
```

Atividade

Em duplas ou individualmente, resolva as seguintes atividades abaixo (todos deverão postar as respostas no moodle):

1) Implemente um código em C capaz de alocar com malloc um vetor de 4 números do tipo inteiro. Depois, o programa deve solicitar ao usuário a entrada dos 4 números no espaço alocado. Por fim, o programa deve mostrar os 4 números e liberar a memória alocada.

2) Implemente um código em C que inicialmente recebe um inteiro (com scanf) que será usado como tamanho de uma string. Depois, o código deve alocar dinamicamente uma string com o tamanho definido, e em seguida, o conteúdo dessa string deve ser preenchido pelo usuário (também com scanf). O programa deve imprimir o conteúdo da string sem seus caracteres numéricos.

3) Implemente um código em C que declare uma struct para o cadastro de trabalhadores de uma empresa. No código:
a) Deverão ser armazenados, para cada trabalhador: cpf, nome e ano de nascimento.
b) Primeiramente, o usuário deverá inserir o número de trabalhadores que serão armazenados
c) O código deverá alocar dinamicamente a quantidade necessária de memória para armazenar os registros dos trabalhadores.
d) O código deverá solicitar a inserção das informações dos trabalhadores
e) Por fim, deverá ser impressa na tela os dados armazenados e deverá ser feita a liberação da memória.

Tipo Abstrato de Dados (TAD) com ponteiros

Tipo Abstrato de dados

O uso de um TAD traz uma série de vantagens:

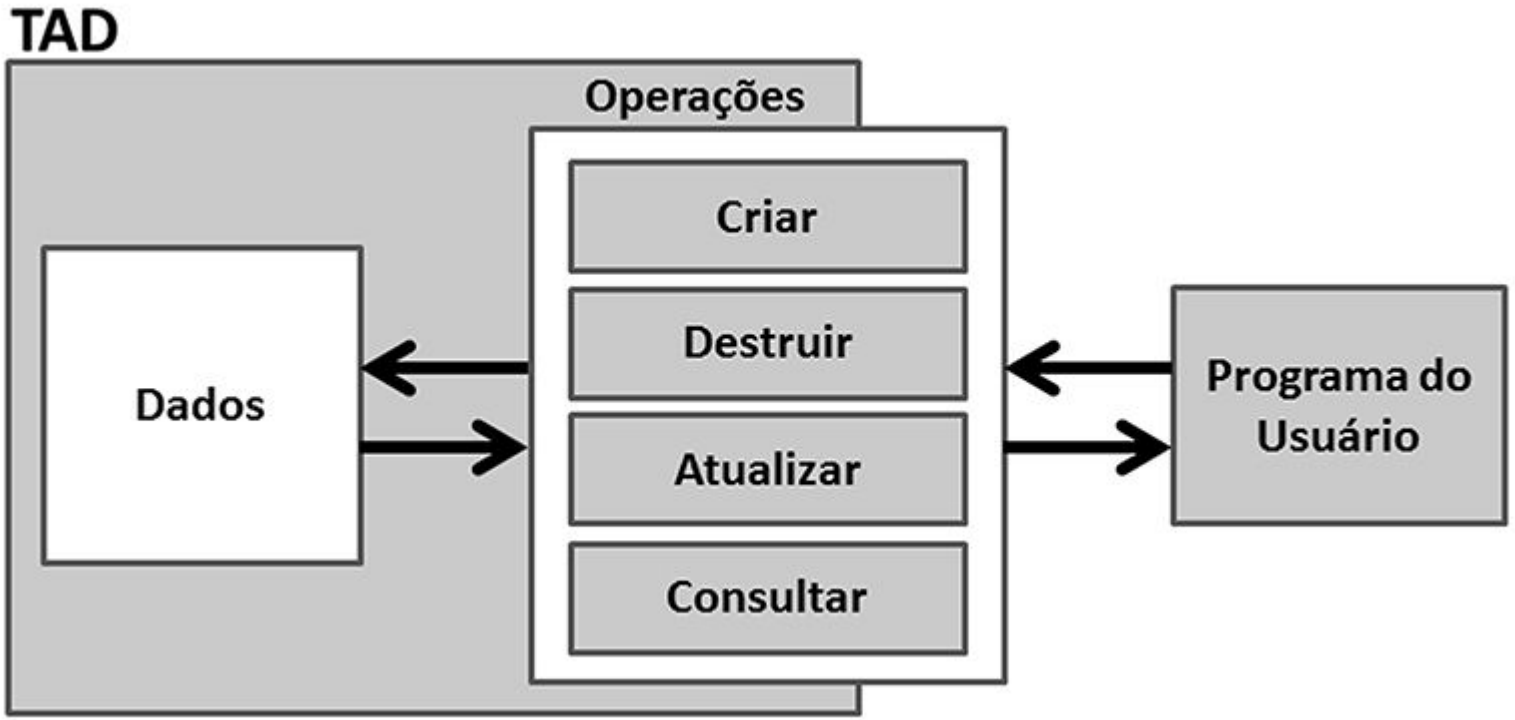
•Encapsulamento: ao ocultarmos a implementação, fornecemos um conjunto de operações possíveis para o TAD.

•Segurança: o usuário não tem acesso direto aos dados. Isso evita que ele manipule os dados de uma maneira imprópria.

•Flexibilidade: podemos alterar o TAD sem alterar as aplicações que o utilizam.

•Reutilização: a implementação do TAD é feita em um módulo diferente do programa do usuário.

Tipo Abstrato de dados



Tipo Abstrato de dados

Exemplo: estrutura do tipo FILE

```
01  typedef struct{
02   int       level;       // nível do buffer
03   unsigned flags;        // flag de status do arquivo
04   char      fd;          // descritor do arquivo
05   unsigned char hold;    // retorna caractere se sem buffer
06   int       bsize;       // tamanho do Buffer
07   unsigned char *buffer; // buffer de transferência de dados
08   unsigned char *curp;   // ponteiro atualmente ativo
09   unsigned      istemp; // indicador de arquivo temporário
10   short      token;      // usado para validação
11  }FILE;
12
```

Alguns acreditam que ninguém, em sã consciência, deve fazer uso direto dos campos dessa estrutura!

Tipo Abstrato de dados

Então, a única maneira de trabalhar com arquivos em linguagem C é declarando um ponteiro de arquivo da seguinte maneira:
FILE* f;
Desse modo, o usuário possui apenas um ponteiro para onde os dados estão armazenados, mas não pode acessá-los diretamente.
A única maneira de acessar o conteúdo do ponteiro FILE é por meio das operações definidas em sua interface.
Assim, os dados do ponteiro f somente podem ser acessados pelas funções de manipulação do tipo FILE:

- fopen()
- fclose()
- fputc()
- fgetc()
- etc

Exemplo de TAD - Ponto

Arquivo Ponto.h

```
01  typedef struct ponto Ponto;
02  //Cria um novo ponto
03  Ponto* Ponto_cria(float x, float y);
04  //Libera um ponto
05  void Ponto_libera(Ponto* p);
06  //Acessa os valores "x" e "y" de um ponto
07  int Ponto_acessa(Ponto* p, float* x, float* y);
08  //Atribui os valores "x" e "y" a um ponto
09  int Ponto_atribui(Ponto* p, float x, float y);
10  //Calcula a distância entre dois pontos
11  float Ponto_distancia(Ponto* p1, Ponto* p2);
```

Arquivo Ponto.c

```
01  #include <stdlib.h>
02  #include <math.h>
03  #include "Ponto.h" //inclui os Protótipos
04  struct ponto{//Definição do tipo de dados
05    float x;
06    float y;
07  };
```

Exemplo de TAD - Ponto

Arquivo Ponto.h

```
01  typedef struct ponto Ponto;
02  //Cria um novo ponto
03  Ponto* Ponto_cria(float x, float y);
04  //Libera um ponto
05  void Ponto_libera(Ponto* p);
06  //Acessa os valores "x" e "y" de um ponto
07  int Ponto_acessa(Ponto* p, float* x, float* y);
08  //Atribui os valores "x" e "y" a um ponto
09  int Ponto_atribui(Ponto* p, float x, float y);
10  //Calcula a distância entre dois pontos
11  float Ponto_distancia(Ponto* p1, Ponto* p2);
```

Arquivo Ponto.c

```
01  #include <stdlib.h>
02  #include <math.h>
03  #include "Ponto.h" //inclui os Protótipos
04  struct ponto{//Definição do tipo de dados
05    float x;
06    float y;
07  };
```

Exemplo de TAD - Ponto

Criando um ponto

```
01  Ponto* Ponto_cria(float x, float y){
02      Ponto* p = (Ponto*) malloc(sizeof(Ponto));
03      if(p != NULL){
04          p->x = x;
05          p->y = y;
06      }
07      return p;
08  }
```

Destruindo um ponto

```
01  void Ponto_libera(Ponto* p){
02      free(p);
03  }
```

Exemplo de TAD - Ponto

Atribuindo um valor ao ponto

```
01  int Ponto_atribui(Ponto* p, float x, float y){
02      if(p == NULL)
03          return 0;
04      p->x = x;
05      p->y = y;
06      return 1;
07  }
```

Calculando a distância entre dois pontos

```
01  float Ponto_distancia(Ponto* p1, Ponto* p2){
02      if(p1 == NULL || p2 == NULL)
03          return -1;
04      float dx = p1->x - p2->x;
05      float dy = p1->y - p2->y;
06      return sqrt(dx * dx + dy * dy);
07  }
```

Exemplo de TAD - Ponto

Exemplo: utilizando o TAD ponto

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include "Ponto.h"
04  int main(){
05      float d;
06      Ponto *p,*q;
07      //Ponto r; //ERRO
08      p = Ponto_cria(10,21);
09      q = Ponto_cria(7,25);
10      //q->x = 2; //ERRO
11      d = Ponto_distancia(p,q);
12      printf("Distancia entre pontos: %f\n",d);
13      Ponto_libera(q);
14      Ponto_libera(p);
15      system("pause");
16      return 0;
17  }
```

Referências

ANDRÉ BACKES. Estrutura de Dados descomplicada em Linguagem C. Elsevier.
(cap 4)
ANDRÉ BACKES. Linguagem C completa e descomplicada. Campus.
(Seção 8.1, 9.2, cap 10 e cap 11)

https://programacaodescomplicada.wordpress.com/indice/linguagem-c/
aulas 35-37, 47-50, 55-65

https://programacaodescomplicada.wordpress.com/2013/03/25/aula-02-modularizacao-e-tad/